REMARKS

Claims 1-8 are currently pending in the present application, of which Claims 1 and 7-8 have been amended.

The term "is utilize" in the Abstract page has been amended to "is utilized." In addition, the word "complimented" on page 12, line 1 has been amended to "complemented." Also, the comma on page 12, line 6 has been amended to a colon, as suggested by the Examiner. Thus, the objection to the specification is believed to be overcome.

The extraneous "and's" in Claim 1 have been deleted. Also, the word "at" in line 20 of Claim 1 has been removed. Thus, the claim objection is believed to be overcome.

In response to the Examiner's request, physical copies of the non-patent documents are attached herein. Also attached is an amended Figure **3** submitted for Examiner's approval.

Rejection under 35 U.S.C. § 112

Claim 7 was rejected under 35 U.S.C. § 112, second paragraph, as being indefinite for not particularly pointing out and distinctly claiming the subject matter that Applicant regards as the invention. Applicant respectfully traverses such rejection insofar as it might apply to the claims as amended herein.

Amended Claim 7 now recites "program code means for changing a branch location in data structures of said SA_Expr3, except for a last data structure of said SA_Expr3, from said second branch location to end of said SA_Expr3" and "program code means for changing said branch location in said last data structure of said SA_Expr3 from a branch to said first location to branch to said second location, or from a branch to said second location to branch to said first location." Because the antecedent basis problems in Claim 7 have been corrected, the § 112 rejection is believed to be overcome.

Rejection under 35 U.S.C. § 102

Claims 1 and 4-5 were rejected under 35 U.S.C. § 102(b) as being anticipated by *Leeper et al.* (Structured Assembly Language in VAX-11 MACRO, Feb. 1986, Proceedings of the 17th SIGCSE Technical Symposium on Computer Science Education, Vol. 18,m issue 1). Applicant respectfully traverses such rejection insofar as it might apply to the claims as amended herein.

Amended Claim 1 now recites program code means for "constructing a data structure referencing said arg1, said arg2, said cc, and a branch destination," "generating a comparison opcode in response to elements of said data structure," "generating a conditional branch based on said condition code in said data structure," "generating a first branch location for execution to proceed as if said structured assembly language expression is true," "generating a second branch location for execution to proceed as if said structured assembly language expression is false," "generating a third branch location for execution to proceed to the end of said structured assembly language expression," and "indicating said branch destination in said data structure is a branch to said first, said second, or said third branch locations." Thus, the above-mentioned steps are related to each other in a coherent fashion.

On pages 5-6 of the Office Action, the Examiner asserts that the above-mentioned inventive steps are disclosed by *Leeper* in various paragraphs on pages 54-57. Assuming *arguendo* that each of the paragraphs of *Leeper* as cited by the Examiner does teach each of the above-mentioned inventive steps, respectively, the paragraphs of *Leeper* are not related to each other in a coherent fashion. For example, the paragraphs on page 54 of *Leeper* are related to an IF-THEN construct, and the paragraphs on page 55 of *Leeper* are related to an IF-THEN-ELSE construct and a FOR LOOP construct. The IF-THEN construct is not related to either the IF-THEN-ELSE construct or the FOR LOOP construct, and there is no coherency among the different concepts. Thus, even if each of the paragraphs of *Leeper* as cited by the Examiner respectively teaches each of the above-mentioned inventive steps, *Leeper* really does not teach the steps of the claimed invention as a whole.

Specifically, *Leeper* also does not teach or suggest some of the claimed steps. For example, Claim 1 recites "program code means for constructing a data structure referencing said arg1, said arg2, said cc, and a branch destination." On page 5 of the Office Action, the Examiner asserts that such program code means is disclosed by *Leeper* in the third paragraph of page 54. However, the IF-THEN construct as shown on page 54 is not a data structure as claimed.

Claim 1 then recites "program code means for generating a comparison opcode in response to elements of said data structure." On page 5 of the Office Action, the Examiner asserts that such program code means is disclosed by *Leeper* in various paragraphs of pages 54-57. However, none of the cited paragraphs of *Leeper* teaches or suggests a comparison opcode generated in response to the elements (*i.e.*, arg1, arg2, cc, and a branch destination) of the data structure, as claimed.

Because the claimed invention recites novel features that are not taught or suggested by *Leeper*, the § 102 rejection is believed to be overcome.

CONCLUSION

Claims 1-8 are currently pending in the present application. For the reasons stated above, Applicant believes that independent Claim 1 along with its dependent claims are in condition for allowance. The remaining prior art cited by the Examiner but not relied upon has been reviewed and is not believed to show or suggest the claimed invention.

No fee or extension of time is believed to be necessary; however, in the event that any addition fee or extension of time is required for the prosecution of the present application, please charge it against IBM Deposit Account No. **50-0563**.

Respectfully submitted,

Antony P. Ng
Registration No. 43,427
DILLON & YUDELL, LLP
8911 N. Capital of Texas Hwy., suite 2110
Austin, Texas 78759
(512) 343-6116

ATTORNEY FOR APPLICANT

AB

Nov. 1991 ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp98-100 -->

Title: # Analysis of Complex Assembler Programs.

Text

TDB

A utility computer program (tool) for analyzing assembler programs is disclosed.

Assembler programs often contain many conditional branches, so that there will be a very large number of different flows of control. Such a program cannot be comprehensively tested within an acceptable time. For example, if there are only 20 independent conditional branches, then over one million executions of the program are required to ensure that every possible path through the program has been tested. In a practical situation many programs contain far more than 20 conditional branches.

Any tool that attempts to perform some kind of flow-of-control analysis may suffer an unacceptable performance for this reason. Thus, practically useful tools need to perform their analysis, or limit their analysis, in some way that does not involve following each and every possible flow of control through the program being analyzed.

An assembler program consists of one or (usually) more FLOW UNITS, where a FLOW UNIT is one or more assembler statements ending with a branch of some kind.

To explain the operation of the analysis tool it is necessary to make some simplifying assumptions. One of these is that control is only passed to a flow unit at determinable place, such as the first statement or at a label. The state changes for each flow unit can then be analyzed.

A flow unit ends with a branch to a flow unit. If the state changes are analyzed through all the flow units in the program being analyzed it is found that the target of a branch may also be the target of another branch. Alternatively, control may also be passed by simple sequential flow of control. In any case, it is possible logically to combine the states as determined by the "from" flow units to determine a worst-case state at a branch target. Assumptions i) While this theory of logically combining states will work, there can be exceptions. Fortunately, these exceptions occur very rarely. An example is when an address is computed from the sum of the contents of two registers, X and Y. Suppose that one path of control sets X to a

valid address and Y to zero while another path sets X to zero and Y to a valid address. The disclosed logic will then falsely flag a possible program check. ii) The distinction between valid and invalid addresses is not always clear, and simplifying assumptions may need to be made. Thus, in, for example, IBM System/370* architecture, absolute addresses 0 to hex FFF are normally reserved by the hardware or for the operating system. Thus, a tool which treats absolute addresses 0 to hex FFF as invalid will give good results when applied to non-operating system code on most occasions. This assumption may cause occasional "false alarms" but these are obvious on inspection. iii) The analyzing tool can identify all possible branches and their targets. If it cannot, then the analysis may still be useful though less than complete. Example

```
Consider Register 5 in the following assembler code:
BALR    13,0          Establish addressability
USING   *,13
*
SR      5,5           Sets R5 to zero
LTR     6,7
BZ      TARGET        Conditional Branch
*
L       5,=A(TARGET) Load R5 with a valid
address
*
TARGET DS      OH
L       9,0(5)        R5 MUST contain a valid
address
```

The first flow unit consists of the SR 5,5 instruction, which sets R5 to zero, followed by the LTR and BZ instructions which do not alter R5. Thus, R5 cannot contain a valid address on completion of the BZ instruction.

The L 5,=A(TARGET) instruction loads R5 with a valid address, but only when it is executed.

The L 9,0(5) instruction will always be executed. However, a program check will occur if R5 contains zero, as it does if the BZ branch to TARGET occurs.

The tool can analyze this situation by logically taking the worst case (i.e., R5 contains an invalid address) from the R5 address validity computed for all the possible paths of control to the instruction being considered. In this case there are just two:

```
After the BZ TARGET instruction, when R5 has an invalid
address.
After the L 5,=A(TARGET) instruction, when R5 has a
valid address.
```

Taking the worst case tells us that this program could program check when executing the L 9,0(5) instruction. *
Trademark of IBM Corp.

Diagrams: none

Diagrams: none
Order/Fcode/Docket: **91A 62842 / / UK8910050**
PubNo=6
: **Nov. 1991** ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp98-100 -->
Privacy | Legal | Gallery | IP Pages | Advertising | FAQ | Contact Us

AC

: Oct. 1991 ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp259-261 -->

Title: # Functional Binary Assembler.

Text

TDB

Logic Sequencers are commonly used in a small system implementation to control the system functions. The Logic Sequencer, in turn, is driven by a set of instructions or microcode. This microcode allows the logic sequencer to send signals to the system's circuits to achieve the designer's goals.

During the design and implementation cycle, the microcode generation has to be flexible enough to accommodate fluctuations of the logic sequencer design. This flexibility has to be paired with a high-level language to help the designer to quickly and easily change the logic sequencer binary code.

This article explains one approach in the implementation of a tool (Functional Binary Assembler) to generate this microcode.

The Functional Binary Assembler is a 'generic' assembler, where every parameter of the binary code generation is controlled by the user from the mnemonics to be used, to the code generated by these mnemonics, to the binary output file originated by the assembler. Features

The Functional Binary Assembler has the following features: Parameters Definition

These parameters define the environment for the assembler: the input source code file, the definition of the binary output file, the format of the listing file and how each mnemonic should be handled. Low-Level Mnemonics Definition

All the mnemonics are user-declarable and -definable. The user creates a file that contains each mnemonic, its bit pattern and how to form the output code written to the output file. High-Level Mnemonics Definition

The user may define mnemonics based on previously defined mnemonics. Through these definitions, bit patterns, or partial bit patterns that are repeatedly used, do not have to be specified for every mnemonic that uses them. Binary Output File Specification

The binary code created by this assembler is also completely user- definable from the bits in every word, to the length of each word, to the length of the binary file, to the addressing scheme used to create the file. Listing Output File Formatting

The configuration of the output file can be

changed to include lines and messages to help the user to understand the output created by this assembler. Multiple Binary Output Files

Besides creating the output code, this assembler optionally will create a binary data set formatted to be down loaded to Personal Computers, or used in the AUSSIM simulator.

This tool needs four user-created data sets to operate: the source file containing the mnemonics to generate the binary code; two mnemonics data sets containing how each mnemonic will be converted; and the parameters data set with the specification of how the binary code will be created. Source Data Set

The source data set contains the system definition for this assembler run, the mnemonics from which the object code will be created, and directives to the assembler. Low-Level Mnemonic Data Set

This data set is created by the user to describe the mnemonics (or source language) that will be in the source file. This data set gives the assembler the information to translate the source code to binary code. High-Level Mnemonic Data Set

The High-Level Mnemonic data set is optional. This data set is created by the user to describe high-level mnemonics that will be in the source file. These high-level mnemonics, or macros, are especially useful when a combination of low-level mnemonic is repeated throughout the source code. Instead of entering the same combination of low-level mnemonics every time they are needed, a macro may be defined which will have the same value as the low-level mnemonics. The definition of these macros is done by the user based on the low-level mnemonic names instead of their values. This is an optional feature and it is not required by the system. Parameter Data Set

The Parameter data set is created by the user to provide the following information:

```
High-Level (Macros) Mnemonic Data Set Name
Low-Level Mnemonic Data Set Name
Byte length of each Machine Instruction
Instruction length of the Object Data Set
Value of the Mask Pattern
Value of the Fill Pattern
Name of this Assembler Run
Data Byte Bit Flip Request
Data Set Address Flip Request
AUSSIM Data Set Generation Request
"Dash" line insertion in listing data set
"PC" Down Load data set request
Warning Messages Print Request
Assembler Messages
```

The assembler gives progress and error messages as it goes through processing the input data sets. These messages help the user in debugging the source code.

Diagrams: none

Diagrams:	none
Order/Fcode/Docket:	**91A 62657 / / AT8890322** **PubNo=5**
:	**Oct. 1991** ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp259-261 -->

AD

May 1994 ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp19-28 -->

Title:

Assembler Macro Implementation

Text

TDB

This document contains drawings, formulas, and/or symbols that will not appear on line. Request hardcopy from ITIRC for complete article.

The solution described in this article grew out of a need to be able to implement an Assembler Language macro which required many subparameters to support a new diagnostic software program. The conventional method of implementation provided by an Assembler Language compiler Macro Facility was found to have the disadvantages of being:

o more expensive in implementation effort than necessary
o more expensive and "unfriendly" in application for the user th
necessary
and instead the present method of implementation as described belo
was chosen which expands the Macro Facility parameter handling
capability.

The task was to implement an Assembler Language macro interface which described the characteristics of a computer program to be analyzed by the new diagnostic software. It was desirable that this macro be a "one step", easy-to-use tool for a system analyst to be able to describe the state, both static and dynamic, of the computer program under investigation, at the location of a given trace point. Also the macro needed to be usable in a customer environment (in customer programs compiled to execute as VSE/POWER user exit routines) and therefore should not impact existing customer programs. The states chosen at the beginning were:

o static states (indicators at the physical code location of the
trace point)
Module Name name of module where trace point located (e.g.,
"IPW$$LR")
Trace Point ID a unique numerical identifier of the trace point
within the module named (e.g., "001")
Text short text to describe the situation functionally
(e.g., "READING INPUT")
Functional Group the "functional group" to which the code
location belonged (e.g., "CARD READER SUPPORT" or
"TAPE READER SUPPORT") (*)

(Miscellaneous:) Various other characteristics:
Entry Point the ENTRY point from a caller
Return Point the RETURN point to a caller
External Call an EXTERNAL component call
External Return an EXTERNAL component return point
o dynamic states (indicators at the time of the trace point execution)
Data Areas the contents of various "data areas" of main storage, (e.g., an input data area or a task control block) (*)
Resources the names of "resources" being used by the program at the trace point (e.g., the input channel and unit being used,"cuu"). (*)
General Registers the contents of the /370 machine general registers (registers G0-G15)
Access Registers the contents of the /370 machine access registers (registers A0-A15)
The Macro Facility of a conventional Assembler Language Compiler c quiet easily allow the specification of most of the above trace po characteristics (states), e.g., defining the trace point descripti macro as "TRACEMAC", one might go further to define some of the ab parameters using the "keyword" macro definition format as thus:

```
TRACEMAC MODULE=IPW$$LR,ID=001,TEXT="READING INPUT",ENTRY=YE
```

Fig. 1 Initial Macro Definition, "Keyword" Format

Or if one used the "positional" macro definition format as thus:

```
TRACEMAC IPW$$LR,001,"READING INPUT",YES
```

Fig. 2 Initial Macro Definition, "Positional" Format

Problems can arise with the specification of a parameter when:
1. the parameter length exceeds the limit of 127 or 255 character IBM Assembler Language compilers allow an individual parameter maximum length, including the parameter name (e.g., "MODULE" and "="(for keyword parameters), of from 127 up to 255 characters (see IBM Manual GC33-4016, page v, Feature 31).
2. the parameter needs to be repeated (the compiler will not allo "keyword" parameter to be repeated, and a conventional "positional" parameter can not be repeated by definition).
Such problems arose with the definition of the trace point definit macro parameters (*):
o the parameter describing the "functional group" to which the c location belonged was allowed to have a very large length to allow the user to
Ä specify long functional group names to increase their expressiveness e.g., "CARD_READER_SUPPORT" or "TAPE_READER_SUPPORT".
Ä specify several functional group names.
A maximum of 127 subparameters each with a maximum length of 24 characters was desired, i.e., the parameter would allow the user to specify 127 x 24 = 3048 characters.
o the "data area" parameter needed to be repeated, since a piece code might be implemented to deal with several main storage locations of data, e.g., I/O buffers, control blocks, "common" data area, etc.
o the "resource" parameter needed to be repeated, since a piece code might be implemented to deal with several "resources", e.g., I/O hardware units, terminal users, tasks, etc.

The conventional Macro Facility solution to the above problems is to implement the macro so that the user repeatedly issues the macro until the necessary parameter subparameters have been expressed and end by having some indicator that the user has issued the macro for the final parameter (here illustrated with the parameter "TYPE=LAST"). As an example of the parameter to describe the "data areas" of some trace point, see Figs. below (note that for the "positional" format that the 6th positional parameter has been chosen for the "data area" parameter).

```
TRACEMAC TYPE=BEGIN,MODULE=IPW$$LR,ID=001,TEXT="READING INPUT"
ENTRY=YES
  TRACEMAC DATA=(......)          DETAILS OF DATA AREA 1
  TRACEMAC DATA=(......)          DETAILS OF DATA AREA 2
  TRACEMAC DATA=(......)          DETAILS OF DATA AREA 3
  TRACEMAC TYPE=LAST              FINISHED WITH DESCRIPTION
```

Fig. 3 Illustration of Conventional Solution, "Keyword" Format

```
  TRACEMAC BEGIN,IPW$$LR,001,"READING INPUT",YES
  TRACEMAC ,,,,,(.......)         DETAILS OF DATA AREA 1
  TRACEMAC ,,,,,(.......)         DETAILS OF DATA AREA 2
  TRACEMAC ,,,,,(.......)         DETAILS OF DATA AREA 3
  TRACEMAC LAST                   FINISHED WITH DESCRIPTION
```

Fig. 4 Illustration of Conventional Solution, "Positional" Form

This has the following disadvantages:

1. prior to "last" macro call, the implementation requires the ma to archive its internal state variables every time it is issued to be able "remember" its situation when it last completed.
2. the above archiving requires "global" archive resources (inste of "local" variables) and has the extra burden that the archived symbols must be uniquely defined with respect to the total of all other macro definitions existing so as not to destroy other macro global variables. This is a problem in a customer environment, since there is the possibility that one might accidentally duplicate a user-defined label and therefore endanger customer data integrity.
3. the implementation requires also that the macro has retrieved from the archive the previous internal state, to be able to resume when called the next time.
4. the user of the macro has to repeatedly issue the macro
5. the user has to issue the macro with some parameter indicating which macro call is the "last" one.

An assembler Macro Facility typically offers two formats of defini a macro parameter:

1. keyword format
2. positional format

The total of macro definition parameters may contain a mix of the different formats, with the positional format parameters preceding the keyword parameters thus:

```
        MACROXXX  pospara1,pospara2, ...  posparan,keypara1=xx
```

```
| keypara2, ...  keyparan=zzz
|
| Fig. 5  Macro Definition Parameter Format
```

For a parameter having multiple "subparameters", these are indicat by enclosing them in parentheses and separated by commas. The following Figure illustrates the "data area" parameter definition using subparameters:

```
| Keyword Parameter with Subparameters:
|
|              DATA=(subpara1,subpara2, ...  subparan)
|
| Positional Parameter with Subparameters:
|
|              (subpara1,subpara2, ...  subparan)
|
| Fig. 6  Example of Subparameter Definition Format
```

The keyword format offers the advantages of:
1. the order of the parameters specified is not important
2. the presence of a parameter is not required
3. the parameter is "self-defining" and therefore easy to read (i.e., MODULE=name indicates that "name" describes a module and not something else)

The keyword format has the disadvantages of:
1. it cannot be repeated (the compiler rejects any but the first specification).

The positional format has the disadvantages of:
1. the order of the parameters specified is important
2. the presence of a parameter is required (its presence is evidenced by the delimiter comma ",")
3. parameters are not "self-defining" and therefore difficult to read
4. conventional positional parameters cannot be repeated.

This article describes a solution which addresses all of the disadvantages mentioned so far.

o it is less expensive in implementation effort than the conventional solution
1. it uses no "last" macro call, therefore does not have to archive its internal state variables
2. it requires no "global" archive resources, therefore can be implemented in isolation and without knowledge of other macro definitions (i.e., improved "software engineering" through encapsulated code)
3. it does not have to retrieve a previous archived internal state

o it is less expensive and "friendlier" in application for the u than the conventional solution
1. the user issues the macro only once.
2. the user does not have to issue the macro with some parameter indicating which macro call is the "last" one.

The solution offers a new parameter format which has the advantages of both of the two existing positional and keyword formats, and the disadvantages of neither.

The following is an illustration of the new parameter format for the "data area" parameter. It is defined to the Macro Facility as a positional parameter with two or more subparameters. The first subparameter defines the parameter "label" as is used for the keyword format. If only the second

subparameter is offered, then it is the parameter value. If the second subparameter is followed by other subparameters, then they together are the subparameter values.

```
New Parameter Format with Parameter Value

              (DATA=,para1)

New Parameter Format with Subparameter Values

              (DATA=,subpara1,subpara2, ...  subparan)
```

Fig. 7 Example of New Subparameter Definition Format

The solution described here will now be illustrated. For this purpose use will be made of newly defined parameters for the trace point macro definition:

- "TGP=" parameter defining trace point "functional group"
- "DATA=" parameter defining "data area" characteristics
- "RES=" parameter defining "resource" characteristics

Using the new parameter format, the trace point definition macro "TRACEMAC" design begun in the earlier Figure will be expanded to include the more difficult problem-causing parameters as follows using the "keyword" macro definition format:

```
    TRACEMAC (TGP=,CARD_READER_SUPPORT),
             (DATA=, ...  ),
             (DATA=, ...  ),
             (RES=,  ...  ),
             (RES=,  ...  ),
             (TGP=,TAPE_READER_SUPPORT,DISKETTE_READER_SUPPORT),
             (DATA=, ...  ),
             MODULE=IPW$$LR,ID=001,TEXT="READING INPUT",ENTRY=YE
```

Fig. 8 Example of New Format Macro Definition Using Keyword Format

f one used the "positional" macro definition format as follows:

```
   TRACEMAC  IPW$$LR,001,"READING INPUT",YES,
             (TGP=,CARD_READER_SUPPORT),
             (DATA=, ...  ),
             (DATA=, ...  ),
             (RES=,  ...  ),
             (RES=,  ...  ),
             (TGP=,TAPE_READER_SUPPORT,DISKETTE_READER_SUPPORT),
             (DATA=, ...  )
```

Fig. 9 Example of New Format Macro Definition Using Positional Format

It should be noted in the above that the "TGP=" parameter, the "DATA=" parameter and the "RES=" parameter are:

- repeated,

o in a mixed order,
o each with its own "label" for easy reading.
combining the advantages of the keyword parameter and the position parameter and eliminating their individual disadvantages.

The above Example uses the macro only once without any "last" indicator, showing it to be user-friendly and with the less user effort over the conventional method.
The implementation method uses the Macro Facility capability to provide the implementer with a system variable giving the:

o total number of positional parameters N'&SYSLIST (see IBM Manu GC33-4016, page 283)
o value of each positional parameter &SYSLIST(n) and subparamete &SYSLIST(n,m) (see IBM Manual GC33-4016, page 281)
In processing one of the new format parameters (e.g., the "DATA=" parameter) the implementation performs a loop over all the positio parameters until it locates one having the first subparameter &SYSLIST(n,1) with the value "DATA=". It then processes the rest the subparameters, and continues searching for another such parame until all are processed. It therefore does not have to archive it internal state after processing the first parameter to process the next and therefore makes no use of a "global" archive.
The solution will now be demonstrated:

Diagrams:

```
TRACEMAC MODULE=IPWSSLR,ID=001,TEXT='READING INPUT',ENTRY=YES
```
Fig. 1 Initial Macro Definition, "Keyword" Format

```
TRACEMAC IPWSSLR,001,'READING INPUT',YES
```
Fig. 2 Initial Macro Definition, "Positional" Format

```
TRACEMAC TYPE=BEGIN,MODULE=IPWSSLR,ID=001,TEXT='READING INPUT',ENTRY=YES
TRACEMAC DATA=(.....)        DETAILS OF DATA AREA 1
TRACEMAC DATA=(.....)        DETAILS OF DATA AREA 2
TRACEMAC DATA=(.....)        DETAILS OF DATA AREA 3
TRACEMAC TYPE=LAST           FINISHED WITH DESCRIPTION
```
Fig. 3 Illustration of Conventional Solution, "Keyword" Format

```
TRACEMAC BEGIN,IPWSSLR,001,'READING INPUT',YES
TRACEMAC ,,,,,(.....)        DETAILS OF DATA AREA 1
TRACEMAC ,,,,,(.....)        DETAILS OF DATA AREA 2
TRACEMAC ,,,,,(.....)        DETAILS OF DATA AREA 3
TRACEMAC LAST                FINISHED WITH DESCRIPTION
```
Fig. 4 Illustration of Conventional Solution, "Positional" Format

```
MACRO00  parm1,parm2, ... sysvar1,keyvar1=xxx,keyvar2=yyy, ... keyvarn=zzz
```
Fig. 5 Macro Definition Parameter Format

```
Keyword Parameter with Subparameters
        DATA=(subparm1,subparm2, ... subparm)
Positional Parameter with Subparameters
        (subparm1,subparm2, ... subparm)
```
Fig. 6 Example of Subparameter Definition Format

```
New Parameter Format with Parameter Value
        (DATA=,parm1)
New Parameter Format with Subparameter Values
        (DATA=,subparm1,subparm2, ... subparm)
```
Fig. 7 Example of New Subparameter Definition Format

```
TRACEMAC (TGP=,CARD_READER_SUPPORT),
         (DATA=, ... ),
         (DATA=, ... ),
         (RES=, ... ),
         (RES=, ... ),
         (TGP=,TAPE_READER_SUPPORT,DISKETTE_READER_SUPPORT),
         (DATA=, ... ),
         MODULE=IPWSSLR,ID=001,TEXT='READING INPUT',ENTRY=YES
```
Fig. 8 Example of New Format Macro Definition Using Keyword Format

```
TRACEMAC  IPWSSLR,881,'READING INPUT',YES,
          (TGP=,CARD_READER_SUPPORT),
          (DATA=, ... ),
          (DATA=, ... ),
          (RES=,  ... ),
          (RES=,  ... ),
          (TGP=,TAPE_READER_SUPPORT,DISKETTE_READER_SUPPORT),
          (DATA=, ... )
```

Fig. 9 Example of New Format Macro Definition Using Positional Format

```
 LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                          XPVM.CMSBZ 10.47 11/25/93
000000                            253 ADCN   CSECT  ,                    ESTABLISH CONTROL SECTION
                                  253        IPWCSECT                    SECTION DESCRIPTION
000000 C9C3C3E3C1C6F8F5           254=C*IPWCSECT ESD  CL16'CSECT SECTION *    SECTION DESCRIPTION
000010                            461 G200   CSECT  ,
                                  463        IPWCOPY  VER=1979,MOD=590  MONITOR COPYRIGHT CONSTANT
000010 4778 F048      0004B       465+       B      *+4+44                BYPASS COPYRIGHT CONST.
000014 F6F9F8F4F6F9F3F7           467+       DC     CL37'5806-033 (C) COPYRIGHT IBM CORP. 1979'
000029 6040F1F9F9F8F040           468+       DC     CL7' 1998 '
                                  470        IPWSAVE ,                    SAVE REGISTERS
000040 90EC D00C      0000C       471+       STM    14,12,12(13)          SAVE CALLER REGISTERS

000044 0640 A0CC      000CC       473        L      04,CAVE               ADDRESS OWN
                            00000 474        USING  CCDS,04               ESTABLISH ADDRESSABILITY
000048 5850 6134      00134       475        L      05,TCCV               ADDRESS QUEUE DESCRIPTOR
                            00004 476        USING  QCDS,05               ESTABLISH ADDRESSABILITY
1
                                  478        TRACEMAC (DATA=,,DSEC_05,X'1B8'),
                                                (DATA=,CAT,816,X'100'),
                                                (RES=,,ADDRAME,05-QNAM-QNDS),
                                                (TGP=,POPFLOAD_RELTYPE, IMPC),
                                                (DEP=,,18,DAME,,8)+1,BEGPPL-1,EOPPCES,X'09'),
                                                ENTRY=YES,
                                                TYPE=CALL
                                  479+* ------ TRACE POINT BEGIN ------------------------------------------------ *
                                  480+*,DEFAULTED TID VALUE SELECTED = '1'
                                  481+*,TXT OPERANDS DEFAULTED. VALUE DENOTED BY='OTHER GO'.
                                  482+* ------ BEGIN OF TRACE CODE ------------------------------------------------*
                                  483=ST10811   EQU *
                      0040C       484+       CLI    CATCCT,CATCCIN1+CATCCNN TRACE INITIAL(IPIAL) OR NORMAL
000040 9F00 A31C      0031C       485+       BNZ    ST30013               ..YES, CONTINUE CHECKING
000054 4770 F05E      0005E       486+       BC     TCTCHKP,DEAD          TASK HAS TRACE WORKAREA
000058 4760 F03A      0003A       487+       BZ     ST30011               ..NO, SKIP RELEASE OF WORKAREA
                      0005C       488=ST30021   EQU *
00005C 9640 0030      00030       489+       STM    R00,R0,TCTCH          STORE TRACE WORKERS'S TO TCB
000060 8700                       490+       CHOP   0,4                   INSURE "DC" ALIGNMENT
000064 4580 F06C      0006C       491+       BAL    04,*+16               LOAD ADDR OF TRACE TPDA AND FLAGS
000068 00000A00                   492=ST10201   DC     A(SPARE1)             ADDRESS OF TRACE POINT TPDA
                                  493=*                                  (IF ABOVE ADDRESS ZERO, THIS IS A "PATCH" TRACE POINT)
00006C 84                         494+       DC     AL1(X'80')            CALLING TASK FLAGS
                                  495=*                                  X'80'    .. YES,POWER PRIVATE TASK/ASSOCIATE
                                  496=*                                  X'40'    .. YES,POWER OF SUBTSK
                                  497=*                                  X'20'    .. YES,POWER MAINTASK
000060 00                         498+       DC     AL1(0)                CALLING TASK SERVICE FLAGS
                                  499=*                                  X'00'    .. IPVDISV OR IPVSRLV NOT ALLOWED
000060 5810 A284      00284       500+       L      R0,CATC               LOAD TRACE MODULE ADDR
000072 47F5 0010      00010       501+       B      16(R5)                BRANCH TO TRACE MODULE
                                  502=*                                  RETURN POINT FROM IPWSTRC
000076 9600 D030      00030       503+       LM     R0,R3,TCTCH           RELOAD TRACE WORKERS'S
2                         00024   504=ST30011   EQU *-ST10811            END
                            0002E 505=ST10811   EQU *-ST10811            LENGTH OF THIS SECTION
                                  506=*------- TRACE POINT END ------------------------------------------------*
```

Fig. 10 Solution Demonstration Figure (1 of 3)

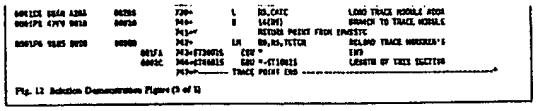
1 Compiled Example of TRACEMAC Macro

```
 LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                          XPVM.CMSBZ 10.47 11/25/93
                                  505 *------------------------------------------------------------------------*
                                  507 *                  SPOOL TAPE PROCESSING  (GET NEXT BLK)                 *
                                  510 *------------------------------------------------------------------------*

                                  601 *       TAPE SPOOLING - READ THE NEXT BLOCK FROM TAPE.

                                  603 *       SINCE THE TAPE MAY HAVE BEEN WRITTEN WITH A DIFFERENT
                                  604 *       BLK- SIZE, THE L. BLOCK IS READ WITH THE MAXIMUM
                                  606 *       DATABLOCK- SIZE SUPPORTED BY VSE/POWER.
                                  608 *       THE RESIDUAL COUNT IS THEN USED TO CALCULATE THE ACTUAL
                                  607 *       DATABLOCK- SIZE.

00F2F2                            607 6E70   GO     R0                                            X22R05W
3                                   610        TRACEMAC (DATA=,T00,R0=(TCTA+TCB2),3,X'40'),
                                                (RES=,,ADDRAME,05-QNAM-QNDS),
                                                TXT='GO-TAPE SPOOLING',
                                                TID=00,
                                                TYPE=CALL
                                  611=* ------ TRACE POINT BEGIN ------------------------------------------------ *
                                  612=* ------ BEGIN OF TRACE CODE ------------------------------------------------*
                            00192 613=ST10913   EQU *
000182 9ACH A31C      0031C       614+       CLI    CATCCT,CATCCIN1+CATCCNN TRACE INITIAL(IPIAL) OR NORMAL
000186 4770 F114      00114       615+       BNZ    ST30013               ..YES, CONTINUE CHECKING
000188 0643 00CC A618 00ACC 00618 616+       CLC    TCTCHKP,ZERO          TASK HAS TRACE WORKAREA?
000118 4780 F12E      0012E       617+       BE     ST30013               ..NO, SKIP RELEASE OF WORKAREA
                            00114 618=ST30413   EQU *
000114 9005 0030      00030       619+       STM    R0,R5,TCTCH           STORE TRACE WORKERS'S TO TCB
000118                            620+       CHOP   0,4                   INSURE "DC" ALIGNMENT
000118 4140 F122      00122       621+       BAL    R4,*+16               LOAD ADDR OF TRACE TPDA AND FLAGS
00011C 00002A0C                   622=ST40IC    DC     A(IPVDSRC)            ADDRESS OF TRACE POINT TPDA
                                  623=*                                  (IF ABOVE ADDRESS ZERO, THIS IS A "PATCH" TRACE POINT)
000120 80                         624+       DC     AL1(X'80')            CALLING TASK FLAGS
                                  625=*                                  X'40'    .. YES,POWER PRIVATE TASK/ASSOCIATE
                                  626=*                                  X'20'    .. YES,POWER OF SUBTSK
                                  627=*                                  X'10'    .. YES,POWER MAINTASK
000121 80                         628+       DC     AL1(0)                CALLING TASK SERVICE FLAGS
                                  629=*                                  X'00'    .. IPVDISV OR IPVSRLV NOT ALLOWED
000122 6660 A298      00298       630+       L      R6,CATC               LOAD TRACE MODULE ADDR
000126 47F5 0410      00410       631+       B      16(R5)                BRANCH TO TRACE MODULE
                                  632=*                                  RETURN POINT FROM IPWSTRC
00012A 9605 D030      00030       633+       LM     R0,R5,TCTCH           RELOAD TRACE WORKERS'S
                            0012E 634=ST30013   EQU *                    END
                            0012C 635=ST40913   EQU *-ST10913            LENGTH OF THIS SEC      Y10N
                                  636=*------- TRACE POINT END ------------------------------------------------*
                                  798 *                  IF THE IMPORTANT FIELDS MATCH THE HEADER CHECK
                                  799 *                  THEN WE HAVE A TRAILER CHEC.

00013C 5810 D170      00170       713        L      R1,TAAV               POINT TO DEB RECORD
000140 0181 9600 1004 00600 01000 712        CLC    ONDISK-TBLPOOFF,0(R1) DOES DEB MATCH CHECK
000144 4770 F10A      0010A       713        BNE    ST74                  ..NO, NOT MULTIVOL

                                  715 *      (WATCH(94) QUEUE BLOCKS HERE;                                 R4,R00W
```

Fig. 11 Solution Demonstration Figure (2 of 3)

2 Compiled Example of TRACEMAC Macro

```
 LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                          XPVM.CMSBZ 10.47 11/25/93
000146 9640 2007      00007       717        SI     TAPE,X'20'            SET EOV TO CLOSE TAPE
4
                                  718        TRACEMAC (DATA=,T00,R0=(TCTA+TCB2),3,X'80'),
                                                (TGP=,POPFLOAD-MULTYOL),
                                                (RES=,,ADDRAME,05-QNAM-QNDS),
                                                TXT='GO-MULTI-VOL TAPE ENV',
                                                TYPE=CALL
                                  719=* ------ TRACE POINT BEGIN ------------------------------------------------ *
                                  720=*,DEFAULTED TID VALUE SELECTED = '2'
                                  721=* ------ BEGIN OF TRACE CODE ------------------------------------------------*
                            0015E 722=ST30015   EQU *
000152 9ACA A31C      0031C       723+       CLI    CATCCT,CATCCIN1+CATCCNN TRACE INITIAL(IPIAL) OR NORMAL
000156 4770 F15A      0015A       724+       BNZ    ST30016               ..YES, CONTINUE CHECKING
000158 0643 00CC A618 00ACC 00618 725+       CLC    TCTCHKP,ZERO          TASK HAS TRACE WORKAREA?
00015C 4780 F15A      0015A       726+       BE     ST30015               ..NO, SKIP RELEASE OF WORKAREA
                            0015E 727=ST30015   EQU *
000160 9605 0030      00030       728+       STM    R0,R5,TCTCH           STORE TRACE WORKERS'S TO TCB
000164                            729+       CHOP   0,4                   INSURE "DC" ALIGNMENT
000168 4140 F152      00152       730+       BAL    R4,*+16               LOAD ADDR OF TRACE TPDA AND FLAGS
00016C 00001AC2                   731=ST40202   DC     A(IPVDSRC)            ADDRESS OF TRACE POINT TPDA
                                  732=*                                  (IF ABOVE ADDRESS ZERO, THIS IS A "PATCH" TRACE POINT)
00016C 10                         733+       DC     AL1(X'80')            CALLING TASK FLAGS
                                  734=*                                  X'80'    .. YES,POWER PRIVATE TASK/ASSOCIATE
                                  735=*                                  X'40'    .. YES,POWER OF SUBTSK
                                  736=*                                  X'20'    .. YES,POWER MAINTASK
000170 00                         737+       DC     AL1(0)                CALLING TASK SERVICE FLAGS
                                  738=*                                  X'00'    .. IPVDISV OR IPVSRLV NOT ALLOWED
```



Compiled Example of TRACEMAC Macro

Order/Fcode/Docket: **94A 61586 / / GE8920141**
Vol=37 PubNo=5
: May 1994 ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp19-28 -->

AE

: Jan. 1979 ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp3356-3357 -->

Title:

General Purpose Assembler For Microprocessors.

Index terms: **Programming JEA**

Text

TDB:

Using macros with existing assemblers for microprocessors presents problems in that either a new assembler has to be written for each new microprocessor or the use of any existing assembler is restricted by the architecture of the original machine for which it was designed. This article describes a General Purpose Assembler (GPA) designed for use with a macro preprocessor and which gives flexible bit handling and error message generation. The assembler is capable of general application and is not restricted to byte orientated architecture. It can be used for microprocessors, micro programming and programmable logic arrays (PLAs).

In order to generate an assembler for a given source language using the GPA. the user must write a series of macros to translate the source language into the powerful low level language understood by the GPA.

The principle of using macros to translate one assembly language into another is not new, but hitherto this approach has always led to problems due to differences in the architectures of the new and old languages. A particular major problem is dealing with error conditions unique to each processor, for example, testing whether branches are in range, etc.

The GPA solves this problem by permitting conditional error messages which can test assemble time values (equated symbols, address counter etc).

The figure shows the general structure of the GPA. The source program may include macros processed by the same processor. The language in which the macros are written depends upon the macro processor, most stand-alone macro processors are suitable (ML/1, GMP could be used).

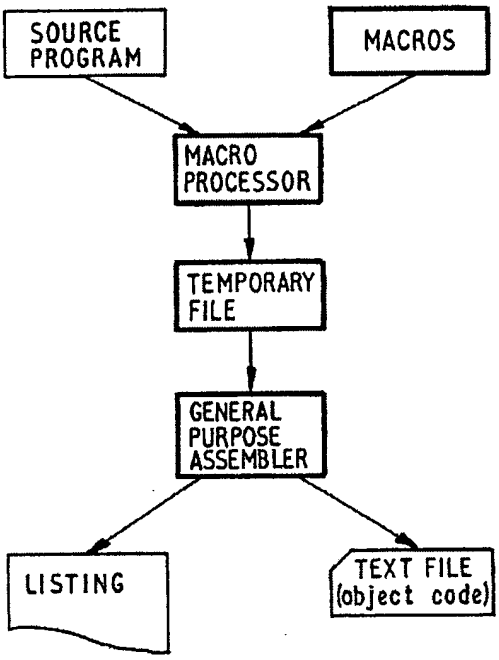
The temporary file holds the original source records, interspersed with lower level GPA statements. (These statements do not appear on the listing.)

The GPA is a single program written in PL/1 and is readily transportable. The GPA

assembles the bit strings required and carries out the necessary housework to produce a useful listing.

The listing is a list of source code with error messages, cross references and error totals.

Diagrams:



Order/Fcode/Docket: **79A 00626 / 75-000. / UK8780091**

: Jan. 1979 ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp3356-3357 -->

AF

: Dec. 1977 ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp2654-2658 -->

Title:

Process to Convert Assembler Language Programs to Decision Tables.

Index terms:

Programming CAK

Text

TDB:

Code development can be accomplished faster and with greater accuracy if the technique of decision tables is applied to the process. A computer-aided method to convert existing Assembler language coding into a decision table format is presented.

In this procedure Assembler language statements provide an input in the form of a matrix, where the number of rows in the matrix correspond to the number of Assembler language statements. To create a decision table, the procedure is used, specifying the name of the matrix containing the Assembler language instructions. The procedure is composed of a number of steps, described below, which analyze the Assembler language instructions and create the decision table.

The first step is to begin creation of an internal table from the matrix of Assembler instructions. The initial internal table will contain the labels, the op codes, the first operand, and the remaining operands of the Assembler input (Fig. 1).

The second step converts "BC Mask, Branch-Label" statements to their extended-mnemonic counterparts, e.g., "BC 8, label" to "BE label". This not only allows easier internal table processing, but also provides easier reading of the code output in the decision table (Fig. 2).

The third step locates all condition setting op codes which are immediately followed by a branch condition instruction and converts them to an internal format for easier processing (Fig. 3).

The fourth step extends the internal table with control fields to control the internal flow of the process in creating the decision table. It does this by sequentially identifying the condition code setting statements and their corresponding branch condition statements for eventual entry into the decision table as conditions. It identifies which statements in the Assembler language program, the "yes" and "no" paths, are to be followed for their respective conditions. It sequentially

identifies all noncondition setting statements (except unconditional branches) as Actions for eventual entry into the decision table. It sequentially identifies all branch register (BR) instructions, all branch (B) instructions to an unidentified label, and all branch instructions to a statement previously encountered in the path currently being processed as Exits, which will eventually be entered into the decision table. It also creates additional fields to be used as forward and backward pointers for tracking the paths associated with the branch condition statements and to indicate whether "yes" or "no" path processing is being performed for a given branch condition statement (Fig. 4).

A fifth step, which is optional, converts "negative" branch condition flow in the internal table to "positive" branch condition flow; i.e., "BN" type branch conditionals are converted to "B" type branch conditionals, while still maintaining the validity of the Assembler program logic. This option eliminates double negative logic in the decision table; i.e., a no for a "BN" type of code is the same as a yes for its corresponding "B" type op code (Fig. 5).

The sixth step creates the condition, Action, and Exit descriptions (or stubs) for the decision table, and initializes their associated rule tables (Fig. 6).

At this point all internal initialization is complete. The seventh step then scans the internal table and makes the appropriate entries in the rule tables. It does this by scanning the internal table following the "yes" and "no" paths of the condition entries in the internal table and entering the related actions with the "Y" or "N" indications in the rule tables until an exit type statement is encountered. Then the appropriate entry is made in the Exit rule table. The basic procedure is to follow the "yes" path for the conditions (entering Ys into the condition rule table for all conditions encountered in the path and the action numbers into the Action rule table for all actions in the path), until an exit type instruction is found.

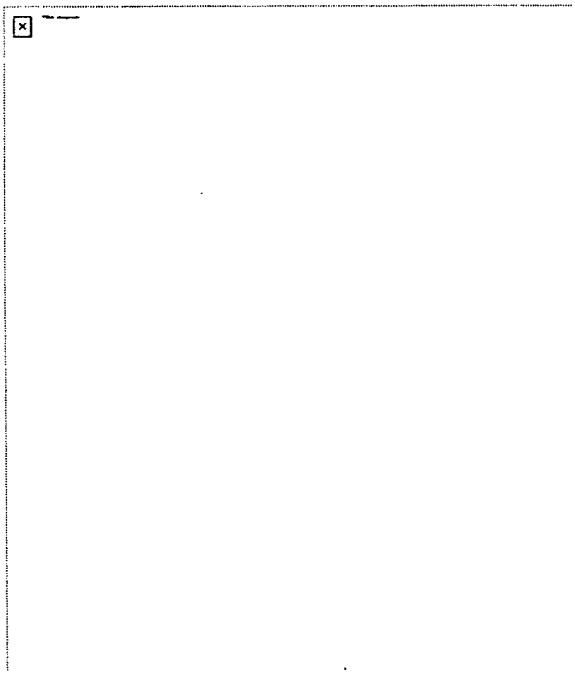
At this point, the last condition is obtained in the internal table and tested to determine whether its "yes" or "no" path was followed. If it indicates that the "yes" path was followed, it is set to indicate that the "no" path is to be followed. All rules in the rule table that were set prior to and including that condition are copied into the next rule column of the rule tables for the conditions and actions, and the rule for the last condition is changed to an N and the "no" path for that condition is followed until an exit is encountered. If the last condition encountered before the exit indicates the "no" path was followed then the backward pointer is obtained which points to the previous condition in the path. The forward and backward pointer and the "yes'" and "no" path indicators are cleared and the path following process is repeated from the previous condition (Fig. 7).

The eighth step locates any unspecified (blank) entries in the condition rule table and converts them to a "don't care" (- dash) indication; i.e., it can be either a Y or N indication (Fig. 8).

At this point the basic parts of the decision table have been completed. The ninth step identifies any actions which have no entries in the Action rule table. This, in effect, is dead code, and probably was never executed in the original Assembler language program (Fig. 9).

The tenth step combines the six basic parts (the stub and rule tables) into the complete decision table. The complete decision table is now a functionally usable statement of the original Assembler language program (Fig. 10).

Diagrams:



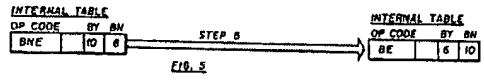


FIG. 5

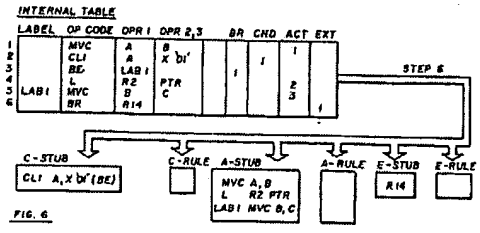


FIG. 6

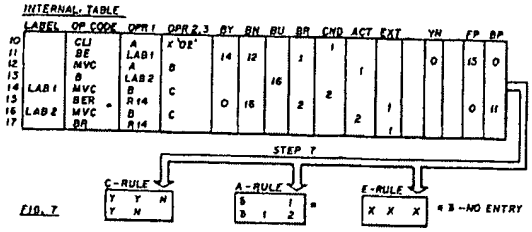
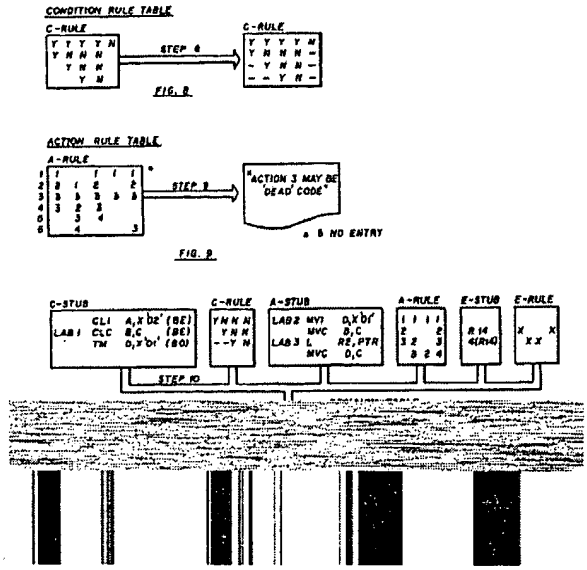


FIG. 7



Order/Fcode/Docket: **77C 02346 / 75-000. / EN8770063**

: Dec. 1977 ....IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp2654-2658 -->

AG

June 1977 ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp368-372 -->

Title: # Fast Assembler using APL.

Index terms: **Programming SMB**

Text

TDB

The typical approach to building an assembler in an APL environment would consist of writing (in APL) subroutines that would read a line of source code, break it up into tokens, store labels in a symbol table, look up op-codes, and so forth. The fast assembly technique involves harnessing these functions already inherent in the APL system itself. By doing this the coding of these functions is avoided and dramatic performance benefits are achieved because these functions (within the APL system) are coded in machine language.

The fast assembly technique takes advantage of a similarity between the syntaxes of APL and assembly language. A line of assembly code can be viewed as a function call. The instruction mnemonic is the name of an APL function. The operand fields of the instruction, separated by commas, are catenated to form the (vector) argument of the function. Thus the APL function ADD would generate the bit pattern for an ADD instruction in the target machine. (Note that throughout this description the names of APL functions and variables will be capitalized and underscored.) With this view, an assembly source program would be an APL function consisting of a series of calls upon ADD and other such "assembly functions". Furthermore, execution of this source program (in the proper context) would actually perform the entire assembly. In essence, this view is the core of the method we will call the "fast assembly" technique.

A sample source program is shown in Fig. 1. This program is both an APL program and an assembly source program for a hypothetical computer. When SOURCE (Fig. 1) is executed, the first line is skipped because it is a comment. The second line invokes the function ENTRY.

```
ENTRY performs the ENTRY assembly function.  The third line perfor
the EXTRN function, the fourth line, the ADD function and so on.
constraining the syntax of the assembly language to conform to tha
of APL we can cause this program to take on a dual function which
allows us to apply the language processing functions of the APL
system to the task of assembly.  As a result, approximately two
```

orders of magnitude in speed improvement can be achieved over codi these language functions in APL.

The data flow for the fast assembly technique is shown in Fig. 2. Pass 1 executes the source program, SOURCE for example, to collect storage allocation information. (Storage allocation information is stored In APL vector Delta LCX. Delta LCX[I] holds a count of the storage required by line I in the source program.) Each function called by SOURCE is capable of operating in each of two modes -- pass 1 mode and pass 2 mode. Because APL line labels have values that are APL line numbers (not related to assembly values), operand fields are ignored during pass 1. (Operand fields of machine instructions typically are not evaluated during pass 1 anyway.) Instead, those instructions ordinarily requiring operand evaluation during pass 1 are deferred. (Their pass 1 action is to place themselves on a deferral list.) After pass 1, but before pass 2, an "interlude" process is carried out. The function of the interlude is to create the pass 2 context for the second execution of SOURCE.

In the pass 2 context all APL labels are redefined to have the assembly-related values determined by pass 1. To do this, the interlude process creates a "context function" and modifies the original source program. Fig. 3 shows the context function for SOURCE. Pass 2 consists of invocation of the context function which establishes the new context, executes the deferred instructions, and finally calls the modified source program. Fig. 4 shows the modified version of SOURCE. In the pass 2 mode all of the functions invoked by SOURCE (modified) generate object code and associated listings. Fig. 5 shows the call topography of the fast assembler. The listings from the sample assembly are shown in Fig. 6.

Because the language functions native to APL need not be explicitly present in the fast assembler, its size is also considerably reduced. Another advantage of the technique is that the source program can be edited with the standard APL function editor. No separate source program editor need be provided.

Diagrams:

FIG. 1

```
     ∇ SOURCE
[1]   ⍝ SAMPLE SOURCE PROGRAM
[2]    ENTRY A4,ERR
[3]    EXTRN'X1,X2'
[4]    ADD NB,NX
[5]   A1:ADDI NB,0
[6]    IF NX,GT,NB,A1
[7]    CGOTO NB,ERR,A1,A2,A3,A4,A5,A6
[8]   ⍝ START OF BRANCH GROUP
[9]   A2:ADD NA,NC
[10]  A3:EQU A2+4
[11]  A4:ADD NB,NX-1
[12]  A5:ADD(A1+1),X2
[13]  A6:ADDI X1,¯5
[14]  ⍝ CONSTANTS
[15]  NB:DC 3
[16]  NA:DS 20
[17]  NX:DC 5
```

```
[18] NC:ORG 100
[19] ERR:DC A4
[20]  END
    ∇
```

FIG. 2

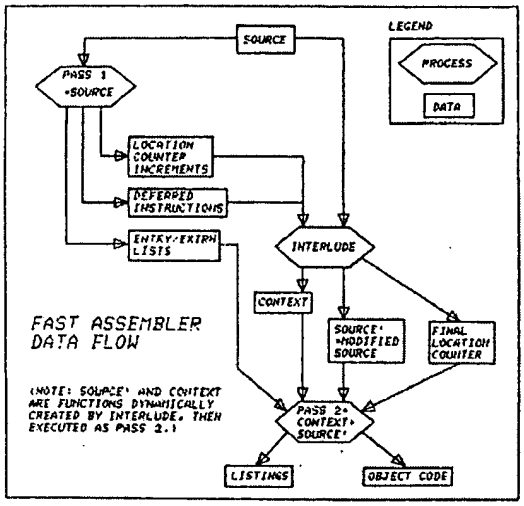
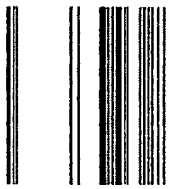


FIG. 6

```
      ASM 'SOURCE '
LOC   OPR  OPND OPND OPND
C--
0000:
0000:
0000:003B 0040 0056
0006:003B 0040 0000
000C: 1057 0056 0040 0006
0014:0056 0040 0006 0064
     0008 002A 002C 002E
     0034 0034
C--
0028:003B 0042 0064
002C:
002E:003B 0040 0055
0034:003A 0007 0001
003A:003B 0000 FFFA
C--
0040:0003
0042:
0056:0005
0064:
0064:002E
0066:

SYMBOL TABLE
 A1        5   6=A 0008
 A2        8  40=A 0028
 A3       10  44=A 002C
```

```
      | SOURCE
 1|A SAMPLE SOURCE PROGRAM
 2|       ENTRY A4,ERR
 3|       EXTRN'X1,X2'
 4|       ADD NB,NX
 5|A1     ADDI NB,0
 6|       IF NX,GT,NB,A1
 7|       CGOTO NB,ERR,A1,A2,A3,A4,A5,A6

 8|A START OF BRANCH GROUP
 9|A2     ADD NA,NC
10|A3     EQU A2+4
11|A4     ADD NB,NX-1
12|A5     ADD(A1+1),X2
13|A6     ADDI X1,-5
14|A CONSTANTS
15|NB     DC 3
16|NA     DS 20
17|NX     DC 5
18|NC     ORG 100
19|ERR    DC A4
20|       END
```



Order/Fcode/Docket: **77C 01175 / 75-000. / SA8760206**

: **June 1977** ... IBM TECHNICAL DISCLOSURE BULLETIN ... <-- pp368-372 -->